# Using Cyclic Memory Allocation to Eliminate Memory Leaks

Huu Hai NGUYEN[1] and **Martin RINARD**[2]
[1]Singapore-MIT Alliance, National University of Singapore
[2]CSAIL, Massachusetts Institute of Technology

***Abstract*— We present and evaluate a new memory management technique for eliminating memory leaks in programs with dynamic memory allocation. This technique observes the execution of the program on a sequence of training inputs to find $m$-bounded allocation sites, which have the property that at any time during the execution of the program, the program only accesses the last $m$ objects allocated at that site. The technique then transforms the program to use *cyclic memory allocation* at that site: it preallocates a buffer containing $m$ objects of the type allocated at that site, with each allocation returning the next object in the buffer. At the end of the buffer the allocations wrap back around to the first object. Cyclic allocation eliminates any memory leak at the allocation site — the total amount of memory required to hold all of the objects ever allocated at the site is simply $m$ times the object size.**

**We evaluate our technique by applying it to several widely-used open source programs. Our results show that it is able to successfully eliminate important memory leaks in these programs. A potential concern is that the estimated bounds $m$ may be too small, causing the program to overlay live objects in memory. Our results indicate that our bounds estimation technique is quite accurate in practice, providing incorrect results for only one of the 161 $m$-bounded sites that it identifies. To further evaluate the potential impact of overlaying live objects, we artificially reduce the bounds at other $m$-bounded sites and observe the resulting behavior. The general pattern that emerges is that overlaying elements of core data structures may make the program fail. Overlaying application data, however, may impair parts of the functionality but does not prevent the program from continuing to execute to correctly deliver the remaining functionality.**

***Index Terms*— Cyclic Memory Allocation, Memory Leak**

## I. Introduction

A program that uses explicit allocation and deallocation has a memory leak when it fails to free objects that it will no longer access in the future. A program that uses garbage collection has a memory leak when it retains references to objects that it will no longer access in the future. Memory leaks are an issue since they can cause the program to consume increasing amounts of memory as it runs. Eventually the program may exhaust the available memory and fail. Memory leaks may therefore be especially problematic for server programs that must execute for long (and in principle unbounded) periods of time.

This paper presents a new memory management technique for eliminating memory leaks. This technique applies to allocation sites [1] that satisfy the following property:

*Definition 1 ($m$-Bounded Access Property):* An allocation site is $m$-bounded if, at any time during the execution of the program, the program only accesses at most the last $m$ objects allocated at that site.

It is possible to use the following memory management scheme for objects allocated at a given $m$-bounded allocation site:

- **Preallocation:** Preallocate a buffer containing $m$ objects of the type allocated at that site.
- **Cyclic Allocation:** Each allocation returns the next object in the buffer, with the allocations cyclically wrapping around to the first object in the buffer after returning the last object in the buffer.
- **No-op Deallocation:** Convert all deallocations of objects allocated at the site into no-ops.

This cyclic memory management scheme has several advantages:

- **No Memory Leaks:** This memory management scheme eliminates any memory leak at allocation sites that use cyclic memory management — the total amount of memory required to hold all of the objects ever allocated at the site is simply $m$ times the object size.
- **Simplicity:** It is extremely simple both to implement and to operate. It can also reduce the programming burden — it eliminates the need for the programmer to write code to explicitly deallocate objects allocated at $m$-bounded allocation sites (if the program uses explicit allocation

*This research was supported in part by DARPA Cooperative Agreement FA 8750-04-2-0254, DARPA Contract 33615-00-C-1692, the Singapore-MIT Alliance, and the NSF Grants CCR-0341620, CCR-0325283, and CCR-0086154.

[1]An allocation site is a location in the program that allocates memory. Examples of allocation sites include calls to `malloc` in C programs and locations that create new objects in Java or C++ programs.

and deallocation) or to track down and eliminate all references to objects that the program will not access in the future (if the program uses garbage collection).

To use cyclic memory management, the memory manager must somehow find $m$-bounded allocation sites and obtain a bound $m$ for each such site. Our implemented technique finds $m$-bounded sites and estimates the bounds $m$ empirically. Specifically, it runs an instrumented version of the program on a sequence of sample inputs and observes, for each allocation site and each input, the bound $m$ observed at that site for that input.[2] If the sequence of observed bounds stabilizes at a value $m$, we assume that the allocation site is $m$-bounded and use cyclic allocation for that site.

One potential concern is that the bound $m$ observed while processing the sample inputs may, in fact, be too small: other executions may access more objects than the last $m$ objects allocated at the site site. In this case the program may overlay two different live objects in the same memory, potentially causing the program to generate unacceptable results or even fail.

To evaluate our technique, we implemented it and applied it to several sizable programs drawn from the open-source software community. We obtained the following results:

- **Memory Leak Elimination:** Several of our programs contain memory leaks at $m$-bounded allocation sites. Our technique is able to identify these sites, apply cyclic memory allocation, and effectively eliminate the memory leak.
- **Accuracy:** We evaluate the accuracy of our empirical bounds estimation approach by running the programs on two sets of inputs: a training set (which is used to estimate the bounds) and a larger validation set (which is used to determine if any of the estimated bounds is too small). Our results show that this approach is quite accurate: the validation runs agree with the training runs on all but one of the 161 sites that the training runs identify as $m$-bounded.
- **Impact of Cyclic Memory Allocation:** In all but one of the programs, the bounds estimates agree with the values observed in the validation runs and the use of cyclic memory allocation has no effect on the observable behavior of the program (other than eliminating memory leaks). For the one program with a single bounds estimation error (and as described further in Section IV-A.3), the resulting overlaying of live objects has the effect of disabling some of the functionality of the affected program. The remaining functionality remains intact; the error does not cause the program to fail or otherwise interfere with its continued operation.
- **Bounds Reduction Effect:** To further explore the potential impact of an incorrect bounds estimation, we artificially reduced the estimated bounds at each site and investigated the effect that this artificial reduction had on the program's behavior. In most of the cases this reduction impaired some of the program's functionality. It did not, however, cause the program to fail and in fact left the program able to execute code that accessed the overlaid objects to continue on to provide the remaining functionality. When the reduction caused the program to fail, the involved objects participated in core data structures with consistency properties that cut across multiple objects. Finally, in some cases the reduction did not impair the observed behavior of the programs at all. An investigation indicates that in these cases, the affected part of the program has been coded to either detect or tolerate inconsistent values.

Our conclusion is that cyclic memory allocation with empirically estimated bounds provides a simple, intriguing alternative to the use of standard memory allocation approaches for $m$-bounded sites. It eliminates the need for the programmer to either explicitly manage allocation and deallocation or to eliminate all references to objects that the program will no longer access. One particularly interesting aspect of the results is the indication that it is possible, in some circumstances, to overlay live objects without unacceptably altering the behavior of the program as long as the core data structures remain consistent.

This paper makes the following contributions:

- **$m$-Bounded Allocation Sites:** It identifies the concept of an $m$-bounded allocation site.
- **Cyclic Memory Allocation:** It proposes the use of cyclic memory allocation for $m$-bounded allocation sites as a mechanism for eliminating memory leaks at those sites.
- **Empirical Bounds Estimation:** It proposes a methodology for empirically estimating the bounds at each allocation site. This methodology consists of instrumenting the program to record the observed bound for an individual execution, then running the program on a range of training inputs to find allocation sites for which the sequence of observed bounds is the same.
- **Experimental Results:** It presents experimental results that characterize how well the technique works on several sizable programs drawn from the open-source software community. The results show that cyclic memory allocation can eliminate memory leaks in these programs and that the programs can, in some circumstances, provide much if not all of the desired functionality even when the bounds are artificially reduced to half of the observed values. One intriguing aspect of these results is the level of resilience that the programs exhibit in the face of overlaid data.

The remainder of the paper is structured as follows. Section II presents an example that illustrates our approach. Section III describes the implementation in detail. Section IV presents our experimental evaluation of the technique. Section V discusses related work. We conclude in Section VI.

## II. Example

Figure 1 presents a (simplified) section of code from the Squid web proxy cache version 2.4.STABLE3 [4]. At line `9` the procedure `snmp_parse` allocates a buffer `bufp` to hold a `Community` identifier. At lines `20` and `21` the

[2] In any single execution, every allocation site has a bound $m$ (which may be, for example, simply the number of objects allocated at that site).

procedure `snmpDecodePacket` writes a reference to the allocated buffer into a structure `checklist` allocated on the stack; at line 27 it writes a reference to the buffer into its parameter `rq`. The procedure `snmpDecodePacket` passes both `checklist` and `rq` on to other procedures. This pattern repeats further down the (transitively) invoked sequence of procedures.

The procedure `snmpDecodePacket` is called by the procedure `snmpHandleUdp`, which passes a pointer to itself as an argument to `CommSetSelect`, which then stores a reference to `snmpHandleUdp` in a global table of structs indexed by socket descriptor numbers. The program then uses the stored reference as a callback.

Any analysis (either manual or automated) of the lifetime of the `bufp` buffer allocated at line 9 in `snmp_parse` would have to track this complex interaction of procedures and data structures to determine the lifetime of the buffer and either insert the appropriate call to `free` or eliminate all the references to the buffer (if the program is using garbage collection). Any such analysis would, at least, need to perform an interprocedural analysis of heap-aliased references in the presence of procedure pointers. In this case the programmer either was unable to or failed to perform this analysis. The program uses explict allocation and deallocation, but (apparently) never deallocates the buffers allocated at this site and therefore contains a memory leak [1].

When we run the instrumented version of Squid on a variety of inputs, the results indicate that the allocation site at line 9 is an $m$-bounded site with the bound $m = 1$ — in other words, the program only accesses the last object allocated at that site. The use of cyclic memory allocation for this site with a buffer size of 1 object eliminates the memory leak and, to the best of our ability to determine, does not harm the correctness of the program. In particular, we have used this version of Squid in our standard computational environment as a proxy cache for several weeks without a single observed problem. During this time Squid successfully served more than 60,000 requests.

```
 1:  u_char *
 2:  snmp_parse(struct snmp_session * session,
 3:             struct snmp_pdu * pdu,
 4:             u_char * data,
 5:             int length)
 6:  {
 7:    int CommunityLen = 128;
 8:
 9:    bufp = (u_char *)xmalloc(CommunityLen+1);
10:    return (bufp);
11:  }
12:  static void
13:  snmpDecodePacket(snmp_request_t * rq)
14:  {
15:    u_char *Community;
16:    aclCheck_t checklist;
17:
18:    Community =
19:      snmp_parse(&Session, PDU, buf, len);
20:    checklist.snmp_community =
21:      (char *) Community;
22:    if (Community)
23:    allow = aclCheckFast(
24:      Config.accessList.snmp, &checklist);
25:    if ((snmp_coexist_V2toV1(PDU))
26:         && (Community) && (allow)) {
27:      rq->community = Community;
28:      snmpConstructReponse(rq);
29:    }
30:  }
31:  void
32:  snmpHandleUdp(int sock, void *not_used)
33:  {
34:    commSetSelect(sock, COMM_SELECT_READ,
35:                  snmpHandleUdp, NULL, 0);
36:    if (len > 0) {
37:      snmpDecodePacket(snmp_rq);
38:    }
39:  }
```

Fig. 1. Memory leak from Squid

## III. Implementation

Our memory management technique contains two components. The first component locates $m$-bounded allocation sites and obtains the bound $m$ for each site. The second component replaces, at each $m$-bounded allocation site, the invocation of the standard allocation procedure (`malloc` in our current implementation) with an invocation to a procedure that implements cyclic memory management for that site. This component also replaces the standard deallocation procedure (`free` in our current implementation) with a modified version that operates correctly in the presence of cyclic memory management by discarding attempts to explicitly deallocate objects allocated in cyclic buffers. It also similarly replaces the standard `realloc` procedure.

### A. Finding $m$-Bounded Allocation Sites

Our technique finds $m$-bounded allocation sites by running an instrumented version of the program on a sequence of training inputs. As the program runs, the instrumentation maintains the following values for each allocation site:

- The number of objects allocated at that site so far in the computation.
- The number of objects allocated at that site that have been deallocated so far in the computation.
- An observed bound $m$, which is a value such that 1) the computation has, at some point, accessed an object allocated at that site $m-1$ allocations before the most recent allocation, and 2) the computation has never accessed any object allocated at that site more than $m-1$ allocations before the most recently allocation.

The instrumentation also records the allocation site, address range, and sequence number for each allocated object. The address range consists of the beginning and ending addresses of the memory that holds the object. The sequence number is the number of objects allocated at that site prior to the allocation of the given object. So, the first object allocated at a given site has sequence number 0, the second sequence number 1, and so on.

The instrumentation uses the Valgrind `addrcheck` tool to obtain the sequence of addresses that the program accesses as it executes []. The instrumentation processes each address and uses the recorded address range information to determine the allocation site and sequence number for the accessed object. It then compares the sequence number of the accessed object with the number of objects allocated at the allocation site so far in the computation and, if necessary, appropriately updates the observed bound $m$.

When the technique finishes running the program on all of the training inputs, it compares the sequence of observed bounds $m$ for each allocation site. If all of the observed bounds are the same for all of the inputs, the technique concludes that the site is $m$-bounded with bound $m$. In this case, the technique generates a production version of the program that uses cyclic allocation for that allocation site with a buffer size of $m$ objects.

### B. Finding Leaking Allocation Sites

Consider an allocation site with an observed bound $m$. If the difference between the number of objects allocated at that site and the number of deallocated objects allocated at that site is larger than $m$, there may be a memory leak at that site. Note that our technique collected enough information to recognize such sites.

It would be possible to use cyclic memory allocation for only such sites. Our current implementation, however, uses cyclic memory allocation for all sites with an observed bound $m$. We adopt this strategy in part because it simplifies the overall memory management of the application and in part because gives us a more thorough evaluation of our technique (since it uses cyclic allocation for more sites).

### C. Implementing Cyclic Memory Management

We have implemented our cyclic memory management algorithm for programs written in C that explicitly allocate and deallocate objects (in accordance with the C semantics, each object is simply a block of memory). Each $m$-bounded allocation site is given a cyclic buffer with enough space for $m$ objects. The allocation procedure simply increments through the buffer returning the next object in line, wrapping back around to the beginning of the buffer after it has allocated the last object in the buffer.

A key issue our implementation must solve is distinguishing references to objects allocated in cyclic buffers from references to objects allocated via the normal allocation and deallocation mechanism. The implementation performs this operation every time the program deallocates an object — the implementation must turn all explicit deallocations of objects allocated at $m$-bounded allocation sites into no-ops, while successfully deallocating objects allocated at other sites. The implementation distinguishes these two kinds of references by recording the starting and ending addresses of each buffer, then comparing the reference in question to these addresses to see if it is within any of the buffers. If so, it is a reference to an object allocated at an $m$-bounded allocation site; otherwise it is not.

### D. Variable-Sized Allocation Sites

Some allocation sites allocate objects of different sizes at different times. We extend our technique to work with these kinds of sites as follows. We first extend our instrumentation technique to record the maximum size of each object allocated at each allocation site. The initial size of the buffer is set to $m$ times this maximum size — the initial assumption is that the sizes observed in the training runs are representative of the sizes that will be observed during the production runs.

At the start of each new allocation, the allocator has a certain amount of memory remaining in the buffer. If the newly allocated object fits in that remaining amount, the allocator places it in the remaining amount, with subsequently allocated objects placed after the newly allocated object (if they fit). If the newly allocated object does not fit in the remaining amount but does fit in the buffer, the allocator places the allocated object at the start of the buffer. Finally, if the newly allocated object does not fit in the buffer, the allocator allocates a new buffer of size $max(2 * m * r, 3 * s)$, where $r$ is the size of the newly allocated object and $s$ is the size of the largest existing buffer for that site.

Note that although applying this extension may result in the allocation of new memory to hold objects allocated at the site, the total amount of memory devoted to the objects allocated at the site is still a linear function of the size of the largest single object allocated at the site, not a function of the number of objects allocated at the site. Because of this bound on the amount of memory allocated at the site, we do not consider the possibility of these allocations to constitute a potential memory leak.

## IV. Evaluation

We evaluate our technique by applying it to several sizable, widely-used programs selected from the open-source software community. These programs include:

- **Squid:** Squid is an open-source, full-featured Web proxy cache [4]. It supports a variety of protocols including HTTP, FTP, and, for management and administration, SNMP. We performed our evaluation with Squid Version 2.4STABLE3, which consists of 104,573 lines of C code.
- **Freeciv:** Freeciv is an interactive multi-player game [2]. It has a server program that maintains the state of the game and a client program that allows players to interact with the game via a graphical user interface. We performed our evaluation with Freeciv version 2.0.0beta1, which consists of 342,542 lines of C code.
- **Pine:** Pine is a widely used email client [3]. It allows users to read mail, forward mail, store mail in different folders, and perform other email related tasks. We performed our evaluation with Pine version 4.61, which consists of 366,358 lines of C code.
- **Xinetd:** Xinetd provides access control, logging, protection against denial of service attacks, and other management of incoming connection requests. We performed our evaluation with Xinetd version 2.3.10, which consists of 23,470 lines of C code.

Note that all of these programs may execute, in principle, for an unbounded amount of time. Squid and Xinetd, in particular, are typically deployed as part of a standard computing environment with no expectation that they should ever terminate. Memory leaks are especially problematic for these kinds of programs since they can affect the ability of the program to execute successfully for long periods of time.

Our evaluation focuses on two issues: the ability of our technique to eliminate memory leaks and on the potential impact of an incorrect estimation of the bounds $m$ at different allocation sites. We perform the following experiments for each program:

- **Training Runs:** We select a sequence of training inputs, typically increasing in size, and run the instrumented version of the program on these inputs to find $m$-bounded allocation sites and to obtain the estimated bounds $m$ for these sites as described in Section III-A.
- **Validation Runs:** We select a sequence of validation inputs. These inputs are different from and larger than the training inputs. We run the instrumented version of the program (both with and without cyclic memory allocation applied at $m$-bounded sites) on these inputs. We use the collected results to determine 1) the accuracy of the estimated bounds from the training runs and 2) the effect of any resulting overlaying of live objects on the behavior of the program.
- **Conflict Runs:** For each $m$-bounded allocation site with $m >$1, we construct a version of the program that uses the bound $\lceil m/2 \rceil$ at that site instead of the bound $m$. We then run this version of the program on the validation inputs. We use the collected results to evaluate the effect of the resulting overlaying of live objects on the behavior of the program.

To evaluate the impact of cyclic memory allocation on any memory leaks, we compare the amount of memory that the original version of the program (the one without cyclic memory allocation) consumes to the amount that the versions with cyclic memory allocation consume.

### A. Squid

Our training inputs for Squid consist of a set of links that we obtained from Google news and a set of SNMP queries that we generated from a Python script that we developed for this purpose. The training inputs have from 157 to 556 links and from 10 to 50 SNMP queries. Our validation input consists of a larger set of links (4290) from Google news and SNMP queries (100) from our Python script. The validation SNMP queries also contain more variables (4) than the training queries (1).

*1) Training and Validation Runs:* Our training runs detected 36 $m$-bounded allocation sites out of a total of 60 allocation sites that executed during the training runs; 28.3% of the memory allocated during the training runs was allocated at $m$-bounded sites. Table I presents a histogram of the observed bounds $m$ for all of the $m$-bounded sites. This table indicates that the vast majority of the observed bounds are small (a pattern that is common across all of our programs). The validation run determines that the observed bound $m$ was too small for 1 out of 36 allocation sites (or 2.8% of the $m$-bounded sites). In this case we say that the validation run *invalidated* these sites.

| m | 1 | 2 | 3 | 14 |
|---|---|---|---|---|
| # sites | 32 | 2 | 1 | 1 |

TABLE I
$m$ DISTRIBUTION FOR SQUID

Table II presents the percentage of executed allocation sites that the training runs identify as $m$-bounded sites, the percentage of memory allocated at these sites, and the percentage of invalidated sites (sites for which the observed bound $m$ was too small) for each of our programs. In general, the training runs identify roughly half of the executed sites as $m$-bounded sites, there is significant amount of memory allocated at those sites, and there are almost no invalidated sites — the training runs deliver observed bounds that are consistent with the bounds observed in the validation runs at all but one of the 161 sites with observed bounds $m$ in the entire set of programs.

| Application | % $m$-bounded | % memory | % invalidated |
|---|---|---|---|
| Squid | 60.0 | 28.3 | 2.8 |
| Freeciv | 50.0 | 75.2 | 0.0 |
| Pine | 63.7 | 36.8 | 0.0 |
| Xinetd | 64.7 | 94.8 | 0.0 |

TABLE II
MEMORY ALLOCATION STATISTICS

*2) Memory Leaks:* Squid has a memory leak in the SNMP module; this memory leak makes squid vulnerable to a denial of service attack [1]. Our training runs indicate that the allocation site involved in the leak is an $m$-bounded site with $m$=1. The use of cyclic allocation for this site eliminates the leak. Figure 2 presents the effect of eliminating the leak. This figure plots Squid's memory consumption as a function of the number of SNMP requests that it processes with and without cyclic memory allocation. As this graph demonstrates, the memory leak causes the memory consumption of the original version to increase linearly with the number of SNMP requests — this version leaks memory every time it processes an SNMP request. In contrast, the memory consumption line for the version with cyclic memory allocation is flat, clearly indicating the elimination of the memory leak.

*3) Effect of Overlaying Live Objects:* Recall that the validation run invalidated one of the $m$-bounded sites. The resulting object overlaying causes Squid to generate incorrect responses to some of the SNMP queries in the validation input. This allocation site allocates SNMP object identifiers, which are then stored in the SNMP MIB tree. This tree is used to look up SNMP variables to respond to SNMP queries. Because the validation input queries a larger number of variables than the training runs, the SNMP object identifiers conflict in the MIB tree. When Squid attempts to look up the data for overwritten

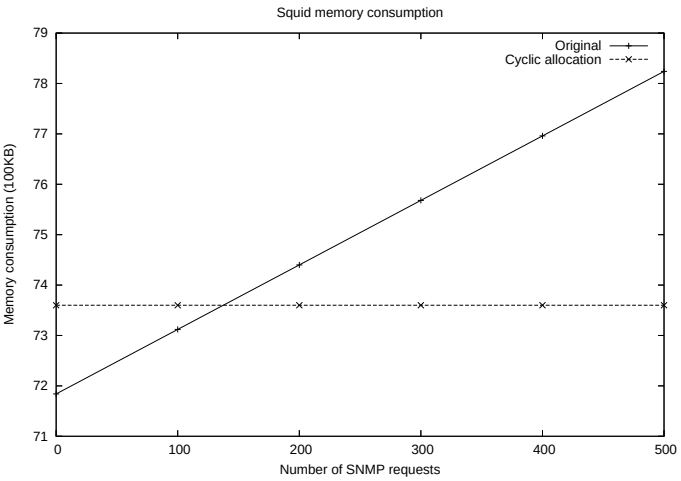


Fig. 2. Squid memory consumption

SNMP object identifiers, it cannot find the data in the tree and returns an empty response. Queries for identifiers that were not overwritten return the correct response. Despite this conflict, this version of Squid continues to execute through the SNMP queries and correctly implements its remaining web proxy functionality.

In hindsight, the identification of the SNMP object identifier allocation site as an $m$-bounded site reflects an inadequacy in the training runs, which all use the same number of SNMP variables. Varying this number of variables in the training runs would cause the observed bounds from the different training runs to vary and the technique would then (correctly) conclude that the site is not $m$-bounded.

*4) Conflict Runs:* For Squid, the training runs find a total of four $m$-bounded allocation sites with $m$ greater than one; one of these sites (the one with $m$=14) is the site discussed above in Section IV-A.3. We next discuss our results from the conflict runs when we artificially reduce the sizes of the observed bounds at the other sites. These results provide additional insight into the potential effect of overlaying live objects in this program.

The first site we consider holds metadata for cached HTTP objects; the metadata and HTTP objects are stored separately. When we reduce the bound $m$ at this site from 3 to 2, the MD5 signature of one of the cached objects is overwritten by the MD5 signature of another cached object. When Squid is asked to return the original cached object, it determines that the MD5 signature is incorrect and refetches the object. The net effect is that some of the time Squid fetches an object even though it has the object available locally; an increased access time is the only potential externally visible effect.

The next site we consider holds the command field for the PDU structure, which controls the action that Squid takes in response to an SNMP query. When we reduce the bound $m$ from 2 to 1, the command field of the structure is overwritten to a value that does not correspond to any valid SNMP query. The procedure that processes the command determines that the command is not valid and returns a null response. The net effect is that Squid is no longer able to respond to any SNMP query at all. Squid still, however, processes all other kinds of requests without any problems at all.

The next site we consider holds the values of some SNMP variables. When we reduce the bound $m$ from 2 to 1, some of these values are overwritten by other values. The net effect is that Squid sometimes returns incorrect values in response to SNMP queries. Squid's ability to process other requests remains completely unimpaired.

### B. Freeciv

Freeciv is designed to allow both human and AI (computer implemented) players to compete in a civilization-building game. Our training inputs for Freeciv consist of from 2 to 30 AI players. The sizes of the game map range from size 4 to size 15 and the games run from 100 to 200 game years. Our validation input consists of 30 AI players and a map size of 20. The game runs for 400 game years.

*1) Training and Validation Runs:* Our training runs detected 42 $m$-bounded allocation sites out of a total of 84 allocation sites that executed during the training runs; 75.2% of the memory allocated during the training runs was allocated at $m$-bounded sites. Table III presents a histogram of the observed bounds $m$ for all of the $m$-bounded sites. As for the other programs, the vast majority of the observed bounds are small. All of the observed bounds in the validation run are consistent with the observed bounds in the training runs; the use of cyclic memory allocation therefore does not change the observable behavior of the program.

| m | 1 | 2 | $> 2$ |
|---|---|---|---|
| # sites | 39 | 3 | 0 |

TABLE III
$m$ DISTRIBUTION FOR FREECIV

*2) Memory Leaks:* It turns out that Freeciv has a memory leak associated with an allocation site repeatedly invoked during the processing of each AI player. Specifically, this allocation site allocates an array of boolean flags that store the presence or absence of threats from the oceans. The training runs determine that this allocation site is an $m$-bounded site with $m$=1. Cyclic memory allocation completely eliminates this memory leak.

*3) Conflict Runs:* Freeciv has three $m$-bounded allocation sites with $m$ greater than 1; all of these sites have $m$=2. All three of these sites are part of the same data structure: a priority queue used to organize the computation associated with path-finding for an AI player. Each priority queue has a header, which in turn points to an array of cells and a corresponding array of cell priorities. The training and validation runs both indicate that, at all three of these sites, the program accesses at most the last two objects allocated. Further investigation reveals that (at any given time) there are at most two live queues: one for cells that have yet to be explored and one for cells that contain something considered to be dangerous. During its execution, however, Freeciv allocates many of these queues.

The first allocation site we consider holds the queue header. Reducing the bound for this site from 2 to 1 causes the size field in the queue header to become inconsistent with the length of the cell and priority arrays. This error causes the program to fail.

Reducing the bounds for the other two sites causes either the cell arrays or the cell priorities to become overlaid. In both cases the program is able to execute successfully without a problem. While the overlaying may affect the actions of the AI players, it is difficult to see this as a serious problem since it does not cause the AI players to violate the rules of the game or visibly degrade the quality of their play.

### C. Pine

Pine is a widely-used email program that allows users to read, forward, and store email messages in folders. Our training inputs have between 3 and 6 mail folders containing between 54 and 141 email messages. Our validation input has 24 mail folders that contain more than 2,500 mail messages.

*1) Training and Validation Runs:* Our training runs detected 72 $m$-bounded allocation sites out of a total of 113 allocation sites that executed during the training runs; 36.8% of the memory allocated during the training runs was allocated at $m$-bounded sites. Table IV presents a histogram of the observed bounds $m$ for all of the $m$-bounded sites. As for the other programs, the vast majority of the observed bounds are small. All of the observed bounds in the validation run are consistent with the observed bounds in the training runs; the use of cyclic memory allocation therefore does not change the observable behavior of the program.

| m | 1 | 2 | 3 | 93 |
|---|---|---|---|---|
| # sites | 68 | 2 | 1 | 1 |

TABLE IV
$m$ DISTRIBUTION FOR PINE

*2) Memory Leaks:* Neither the training nor validation runs revealed a memory leak in Pine.

*3) Conflict Runs:* Pine has four $m$-bounded allocation sites with $m$ greater than 1. The first site we consider has a bound $m$=93. The objects allocated at this allocation site are used to hold mailcap information (this information defines how MIME-encoded content is displayed). When we reduce the bound $m$ at this site from 93 to 47, some of the mailcap information is overwritten. The effect is that Pine is unable to launch some external viewers. In particular, it is unable to launch a web browser to view external content. Even after attempting (unsuccessfully) to launch the web browser, Pine is able to continue to successfully perform other email-related tasks such as reading, forwarding, and viewing content from the mail messages.

The next site we consider has a bound $m$=3. This site allocates nodes in a circular doubly-linked list; each node points to memory holding the text in a Pine status line message. Reducing the bound $m$ from 3 to 2 causes the list pointers to become inconsistent. The effect is that Pine deallocates the text memory twice. Because the text objects are not allocated at an $m$-bounded site, these double deallocations cause Pine to fail.

The final two sites both have bound $m$=2 and are both involved in a list of content filters that convert special characters for display. Reducing the bound $m$ from 2 to 1 causes the list to lose one of the filters. Although this loss did not affect our validation run, it is possible that messages containing characters destined for the lost filter would be displayed incorrectly. However, even after the resulting overlaying of objects, the data structures remain consistent and do not interfere with the successful delivery of other Pine functionality.

### D. Xinetd

Our training inputs for Xinetd consist of between 10 and 500 requests. Our validation input consists of 1000 requests. All of these requests are generated by a Perl script we developed for this purpose.

*1) Training and Validation Runs:* Our training runs detected 11 $m$-bounded allocation sites out of a total of 17 allocation sites that executed during the training runs; 94.8% of the memory allocated during the training runs was allocated at $m$-bounded sites. Table V presents a histogram of the observed bounds $m$ for all of the $m$-bounded sites. All of the observed bounds $m$ are 1. All of the observed bounds in the validation run are consistent with the observed bounds in the training runs; the use of cyclic memory allocation therefore does not change the observable behavior of the program.

| m | 1 | $\geq 2$ |
|---|---|---|
| # sites | 11 | 0 |

TABLE V
$m$ DISTRIBUTION FOR XINETD

*2) Memory Leaks:* Xinetd has a leak in the connection-handling code — whenever Xinetd rejects a connection (it is always possible for an attacker to generate connection requests that Xinetd rejects), it leaks a connection structure 144 bytes long. Our training runs indicate that the allocation site involved in the leak is an $m$-bounded site with $m$=1. The use of cyclic allocation for this site eliminates the leak. Figure 3 presents the effect of eliminating the leak. This figure plots Xinetd's memory consumption as a function of the number of rejected requests with and without cyclic memory allocation. As this graph demonstrates, the memory leak causes the memory consumption of the original version to increase linearly with the number of rejected requests. In contrast, the memory consumption line for the version with cyclic memory allocation is flat, clearly indicating the elimination of the memory leak.

Note that because none of the $m$-bounded allocation sites in Xinetd have $m$ greater than one, we do not investigate the effect of reducing the bounds.

### E. Discussion

Memory leaks are an insidious problem — they are difficult to find and (as the discussion in Section II illustrates) can

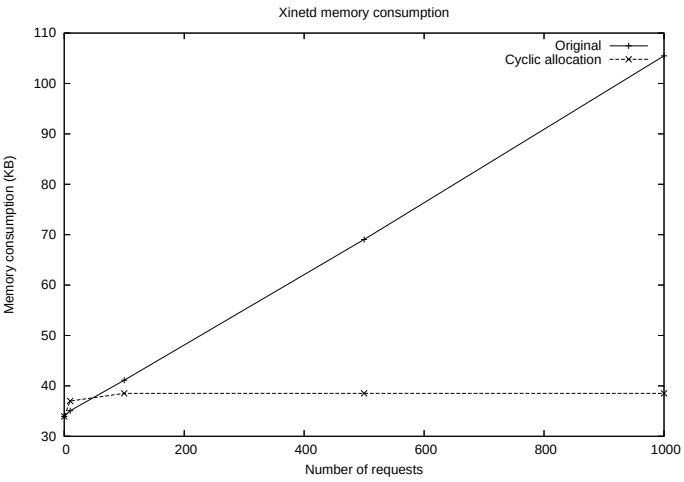


Fig. 3. Xinetd memory consumption

be difficult to eliminate even when the programmer is aware of their presence. Our experience with our four programs underscores the difficulty of eliminating memory leaks — despite the fact that all of these programs are widely used, and in some cases, crucial, parts of open-source computing environments, three of the four programs contain memory leaks.

Our results indicate that cyclic memory allocation enabled by empirically determined bounds $m$ can play an important role in eliminating memory leaks. Our results show that this technique eliminates a memory leak in three of our four programs. If the bounds $m$ are accurate, there is simply no reason not to use this technique — it is simple, easy to implement, and provides a hard bound on the amount of memory required to store the objects allocated at $m$-bounded sites. In this situation there are two key questions: 1) how accurate are the observed bounds, and 2) what are the consequences if the observed bounds are wrong?

Our results indicate that the observed bounds are impressively accurate — the validation inputs invalidate only one of the 161 $m$-bounded allocation sites. Moreover, our conflict runs indicate that the programs are often able to live with the overlaying of live data to continue to execute to successfully deliver much of their functionality to their users. Of the 11 sites considered in the conflict runs, only 2 (one in Freeciv and one in Pine) cause the program to fail if the bound is artificially reduced. Artificially reducing the bounds at 6 of the remaining sites impairs the functionality of part of the program, but leaves the functionality of the other parts intact. Artificially reducing the bounds at the remaining 3 sites leaves the entire functionality intact!

One aspect of our implementation that tends to ameliorate the negative effects of overlaying objects is the fact that different $m$-bounded allocation sites have different buffers. So even if one object overwrites another, the objects sharing the memory will tend to have the same basic data layout and satisfy the same invariants. This property makes the program less likely to encounter a completely unexpected collection of data values when it accesses data from an overwriting object instead of the expected object. This is especially true for application data, in which the values for each conceptual data unit tend to be stored in a single object, with the values in multiple objects largely if not completely independent. For five of the four allocation sites whose reductions cause functionality impairment, overlaying the objects allocated at those sites causes the program to lose the data required to implement the full functionality but does not harm the ability of the program to execute code that accesses the overlaid objects without failure. The program can therefore execute through this code without failing, preserving its ability to deliver other functionality.

This property is, however, much less true for the core data structures, which tend to have important properties that cross object boundaries. And indeed, the two allocation sites whose artificial reductions cause the program to fail both allocate objects that are involved in the core data structures. Moreover, the causes of both program failures stem from inconsistencies that involve multiple objects. In the case of Freeciv, the object overlay causes a length field stored in one object to incorrectly reflect the length of another object. In the case of Pine, the object overlay causes two conceptually distinct list nodes to refer to the same object.

Interestingly enough, for the three cases in which reduction has no effect on the observable behavior, the program is actually set up to tolerate inconsistent values in objects. In one program (Squid) the program anticipates the possibility of inconsistent data and contains code to handle that case. In the other program (Freeciv) the program is able to successfully execute with a range of data values. These two examples suggest that many programs may already have some built-in capacity to fully tolerate inconsistent objects.

## V. Related Work

Dynamic memory management has been a key issue in computer science since the inception of the field. Some languages (C, C++) rely on the programmer to explicitly allocate and deallocate memory. One potential drawback of this approach is the possibility of dangling references — the program may deallocate an object, retain a pointer to the object, then use the retained pointer to access the memory after it has been recycled to hold another object. This memory management approach also leaves the program open to memory leaks if it fails to deallocate objects that it will no longer access in the future.

Garbage collection [13], [10] eliminates the possibility of dangling references by refusing to deallocate any reachable object. The potential drawback is that the program may have a memory leak if it retains references to objects that it will no longer access.

Several researchers have recently developed static program analyses that attempt to find memory leaks and/or accesses via dangling references. Heine and Lam use synthesized ownership properties to discover leaks and multiple deallocations [11]; Hackett and Rugina use an efficient shape analysis to detect memory leaks in programs with explicit deallocation [9]. Shaham, Yahav, Kolodner, and Sagiv use a

shape analysis to eliminate memory leaks in garbage-collected Java programs; the idea is to use static analysis to find and eliminate references that the program will no longer use [12].

All of these techniques entail the use of a heavyweight static analysis. Because the analyses are conservative, they may miss some leaks. Moreover, once the analysis finds the leak, it is the responsibility of the developer to understand and eliminate the leak.

Gheorghioiu, Salcianu, and Rinard present a static analysis for finding allocation sites that have the property that at most one object allocated at that site is live during any point in the computation [8]. The compiler the applies a transformation that preallocates a single block of memory to hold all objects allocated at that site. Potential implications of the technique include the elimination of any memory leaks at such sites, simpler memory management, and a reduction in the difficulty of computing the amount of memory required to run the program. This analysis can be viewed as the static counterpart of our dynamic techniques that determine an observed bound $m$, but with the additional restriction that the bound $m$ equal 1. An advantage of this analysis is that it is sound (i.e., the analysis considers all possible executions and guarantees that no execution will ever have more than one object from the site live at any time); disadvantages include the need to develop a sophisticated program analysis and the potential for the analysis to conservatively miss allocation sites with at most one live object.

Some memory management mechanisms enable the program to overlay live data. A program with explicit deallocation can deallocate an object too early, then use the resulting dangling reference to access the storage that the deallocated object occupied. If the memory manager has allocated another object into this storage, there are, in effect, multiple objects occupying the same storage. Providing a separate allocation pool for each class of objects can preserve type safety even in the face of premature deallocation and the resulting dangling references [7].

Because our technique uses a separate buffer for each allocation site, the resulting memory management algorithm will typically preserve basic type safety even when the system overlays live objects (although, strictly speaking, unsafe C constructs such as unions can cause overlaying to generate additional type safety violations). The remaining key cause of program failure is corruption of the core linked data structures (overlaying general application data usually has much less severe consequences). This property suggests that it may be possible to use data structure repair [5], [6] to make the program substantially more robust in the face of this particular form of corruption.

## VI. Conclusion

Memory leaks are an important source of program failures, especially for programs such as servers that must execute for long periods of time. Our cyclic memory allocation technique observes the execution of the program to find $m$-bounded allocation sites, which have the useful property that the program only accesses at most the last $m$ objects allocated at that site. It then exploits this property to preallocate a buffer of $m$ objects and cyclically allocate objects out of this buffer. This technique caps the total amount of memory required to store objects allocated at that site at $m$ times the size of the objects allocated at that site. Our results show that this technique can eliminate important memory leaks in long-running server programs.

One potential concern is the possibility of overlaying live objects in the same memory. Our results indicate that our bounds estimation technique is very acccurate, misclassifying only one of the 161 allocation sites it determines to be $m$-bounded. Moreover, our results also show that the effect of overlaying live objects is substantially less severe than might be expected. Specifically, overlaying application data may disable part of the functionality of the program, but usually does not cause the program to fail. Our results therefore indicate that cyclic memory allocation, enabled by empirical determination of the bounds $m$, may provide a useful alternative memory management technique to more standard techniques, which remain vulnerable to memory leaks.

## References


[1] CVE-2002-0069. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0069.

[2] Freeciv website. http://www.freeciv.org/.

[3] Pine website. http://www.washington.edu/pine/.

[4] Squid Web Proxy Cache website. http://www.squid-cache.org/.

[5] Brian Demsky and Martin Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '03)*, October 2003.

[6] Brian Demsky and Martin Rinard. Data Structure Repair Using Goal-Directed Reasoning. In *Proceedings of the 2005 International Conference on Software Engineering*, May 2005.

[7] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, June 2003.

[8] Ovidiu Gheorghioiu, Alexandru Salcianu, and Martin Rinard. Interprocedural compatability analysis for static object preallocation. In *Proceedings of the 30th Annual ACM Symposium on Principles of Programming Languages*, January 2003.

[9] Brian Hackett and Radu Rugina. Region-based shape analysis with tracked locations. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 310–323. ACM Press, 2005.

[10] Hans-J. Boehm and Mark Weiser. Garbage collection in an uncooperative environment. In *Software – Practice and Experience*, number 9, pages 807–820, 1988.

[11] David L. Heine and Monica S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 168–181. ACM Press, 2003.

[12] Ran Shaham, Eran Yahav, Elliot K. Kolodner, and Mooly Sagiv. Establishing Local Temporal Heap Safety Properties with Applications to Compile-Time Memory Management. In *The 10th Annual International Static Analysis Symposium (SAS '03)*, June 2003.

[13] Richard Jones and Rafael Lins. *Garbage Collection : Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.